Explorations of Python-Based Automatic Hardware Code Generation for Neutron Transport Applications

Jackson P. Morgan,[1,2,†] Todd Palmer,[2,3] and Kyle E. Niemeyer[1,2]

[1]*School of Mechanical, Industrial, and Manufacturing Engineering, Oregon State University, Corvallis, OR 97331*
[2]*Center for Exascale Monte Carlo Neutron Transport (CEMeNT)*
[3]*School of Nuclear Science and Engineering, Oregon State University, Corvallis, OR 97331*
[†]*Corresponding Author, morgjack@oregonstate.edu*

INTRODUCTION

Monte Carlo / Dynamic Code (MC/DC) is a soon-to-be released Python-based neutron transport solver, developed as part of CEMeNT (Center for Exascale Monte Carlo Neutron Transport). It is a research code used to investigate novel methods for the development of dynamic simulations. As Monte Carlo (MC) neutron transport applications are often highly computationally taxing, a Python-based technique to rapidly gain parallelism for both GPU and CPU hardware is required so methods implemented in MC/DC can be tested at the high performance computing (HPC) scale.

Our objective is to find a technique that presents the best software engineering solution to enable rapid prototyping for methods research in MC/DC at the HPC scale using both CPUs and GPUs. We are not looking to alter current development dynamics for production HPC codes or suggest that these techniques will enable Python-based development—or should be pursued—for all production codes. Instead, we seek to examine production techniques for rapidly developing novel methods at large scales with the ability to take advantage of accelerator hardware.

We implemented core components of MC/DC in a testbed to allow for even faster exploration of parallelism in a Python framework. This testbed, MC/DC-Toy Neutronics Testbed (MC/DC-TNT), is a transient, event-based, mono-energetic solver that enables straightforward parallelization of its compute functions. While the performance of large-scale MC codes is generally memory-bound rather than compute-bound [1], our initial efforts examine techniques related to the latter to gain a foothold in Python-based HPC.

METHODS OF PARALLELIZATION

We examine three methods of parallelization within a Python framework. All three operate by using Python as "glue" language (a language which is used to run other compiled kernels) with a just-in-time (JIT) compilation scheme to produce and run compute kernels. All three are also coupled with automatic code translation protocols, to aid in the rapid development of these kernels, either with the source being written in full Python or some Python-esq form.

PyKokkos

PyKokkos [2] is a Python library implementation for the C++ Kokkos HPC portability model developed by Sandia National Laboratories [3]. PyKokkos is under active development and does not implement all the functionality of Kokkos, nor is it as mature as some of the other methods presented here. PyKokkos currently only implements CUDA and OpenMP back-ends.

More than any other method we consider, PyKokkos has the ability to abstract away the production of compute kernels from their target hardware. It even has the ability to abstract data structures from the user, appearing almost as type-annotated NumPy [4] arrays. While the programming model is entirely Python-based (the user only programs in Python) it can be non-intuitive, as it is based on a paradigm specifically designed for abstraction.

Numba + CUDA

Numba [5] is a translator and compiler for Python that implements the LLVM high performance compiler library, and promises "C-like speeds" for functions that implement it. In simple cases, nothing more than a compiler flag is required to produce a C kernel for a given function that is automatically bound and run by Python. This can even apply to simple functions to be parallelized on a CPU, where parallelization can be implemented with one flag option above a function. However, when functions become more complex, the use of more Numba commands in the Python code is required. It directly supports Nvidia GPUs and seems to be the method of choice for interfacing with the CUDA API from Python.

Recently a sub-module to Numba has been developed called `pyomp` [6] that seeks to allow Numba kernels to take advantage of the OpenMP API for CPU threading. We explore this sub-module and Numba's native "threading" techniques separately from one another as they require slightly different implementations.

While Numba does have some abstractions for specific GPU operations, most of these cannot be leveraged to offload the computational burden in our simulations. In our use case the user will effectively have to write CUDA C kernels within Python. Thus, the user will have to be aware of the hardware architecture, hampering rapid prototyping. Also, Oden found that Numba does include a significant computational overhead, reaching 50–85% of the performance of pure CUDA C versions of compute-intensive benchmarks [7]. In spite of this, we include it in this comparison due to its increasing popularity, ease of use for CPU implementations, and industry support.

Mako Templating Engine

This method has been implemented by the PyFR computational fluid dynamics solver [8] and uses the Mako template engine [9] to abstract and simplify the scripting of compute kernels.

At run time, this approach pushes static kernel templates

through the template engine to reform the kernel into a requisite secondary language for a given hardware API. Then, code is compiled to machine code and bound to Python using a hardware code-generating library.

This type of abstraction, while somewhat convoluted, has enabled PyFR to reach the petascale while being able to run on CPUs and AMD/Nivdia/Intel GPUs [10]. The developers of PyFR have reported less than 1% of computational overhead due to the Python interpreter in this method [11].

We are confident this method will work in parallelizing MC/DC-TNT at HPC scale; however, the initial development needed appears to be significant. While the use of templates does simplify the scripting of compute kernels, architecture-specific knowledge is still required, again, hampering rapid prototyping.

PRELIMINARY FINDINGS

Work is ongoing to implement all methods. Currently Numba (threading and PyOmp) and Pykokkos (OpenMP) are implemented on MC/DC-TNT. Table I shows run time data for the `Advance` kernel (the most compute-intensive function, which implements a surface tracking algorithm), as well as a full integration test for a homogeneous slab problem with vacuum bounds on either side (L=1 cm, $\Delta x = 0.01$ cm, $\Sigma_a = \Sigma_f = \Sigma_s = 0.333\,\mathrm{cm}^{-1}$,).

All techniques demonstrate the viability of these parallelization methods with a significant decreases in run time between pure Python and any parallelization technique. As some kernels must be implemented in serial, an integration test for a slab problem shows all of the methods' parallelization abilities as well as the serial speed benefits of JITed code when compared to a pure Python implementation. PyKokkos does seem to be more performant than both Numba implementations, with Numba PyOmp being less performant than Numba threading. PyOmp has not yet been incorporated into the main distribution of Numba and as such does not receive the continual updates that Numba threading does. We expect that, when it is, we will see this gap between performance of the two implementations narrow.

JIT compilation methods require an initial compilation step when they are being run for the first time on a given system with a given code (any alteration to kernel source code will require this to be redone). Often these compiled binaries are cached in order to prevent this extra runtime hit when ran subsequently. However, depending on problem size, the compilation time can be a significant percentage of overall runtime and should be considered. Table II shows overall compilation time when "warming up kernels" (making an initial call with dummy values) before the simulation starts, per method and hardware target. PyKokkos OpenMP takes 7 times longer than the Numba implementations to translate and compile the kernels though all methods finish a matter of seconds.

We also consider the difficulty of these implementations when coming from a pure Python code. The Numba CPU implementation was easy for simpler kernels, functioning as advertised with a single compiler flag. More complex kernels, especially ones designed to be run in parallel, had to be reformed and use in kernel numba commands to work. Also, we found that when Numba wasn't able to produce kernels for any reason it's error messages directing us to the issues where either cryptic or nonexistent. We found working within Pykokkos to be more difficult than Numba for the CPU implementation. Fueled again by cryptic runtime errors, frustrating type casting issues, and a lack of documentation both for the build process as well as module commands. We expect as PyKokkos matures these issues will be alleviated. It should also be stated the allure of PyKokkos is not only a slight increase of performance on CPU implementations when compared to Numba but also in its portability, with promises of CUDA GPU implementations with changing a single variable. Early implementations of PyKokkos CUDA on MC/DC-TNT does seem to confirm this.

TABLE I. Run time of `Advance` kernel and integration test in seconds, using 1×10^8 particles and 16 threads (if method is parallelized); kernel compilation time not included.

Method	Advance	Integration
Pure Python (CPU)	5.140×10^4	5.297×10^4
Numba (Threading)	1.887×10^2	2.323×10^2
Numba (PyOmp)	2.876×10^2	3.825×10^2
PyKokkos (OpenMP)	1.480×10^2	1.548×10^2

TABLE II. First run translation and compilation wall-clock times for all kernels, in seconds.

Method	Compilation time
Numba (Threading)	4.99
Numba (PyOmp)	5.66
PyKokkos (OpenMP)	37.50
PyKokkos (CUDA)	39.72

FUTURE WORK

Significant work remains before we can select a method to implement in MC/DC. Our immediate goals are to complete the PyKokkos CUDA and Numba CUDA implementations before finally moving onto the Mako templating engine method. We will also implement the AZURV1 [12] transient benchmark to further validate MC/DC-TNT as well as examine performance under different nuclear material data regimes.

ACKNOWLEDGMENTS

The authors acknowledge useful discussions with Dr. Ilham Variansyah.

This work was supported by the Center for Exascale Monte-Carlo Neutron Transport (CEMeNT) a PSAAP-III project funded by the Department of Energy, grant number: DE-NA003967.

REFERENCES

1. J. R. TRAMM and A. R. SIEGEL, "Memory bottlenecks and memory contention in multi-core Monte Carlo trans-

port codes," *Annals of Nuclear Energy*, **82**, 195–202 (2015).
2. N. A. AWAR, S. ZHU, G. BIROS, and M. GLIGORIC, "A performance portability framework for python," in "Proceedings of the International Conference on Supercomputing," Association for Computing Machinery (6 2021), pp. 467–478.
3. H. CARTER EDWARDS, C. R. TROTT, and D. SUNDERLAND, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, **74**, *12*, 3202–3216 (2014).
4. C. R. HARRIS, K. J. MILLMAN, S. J. VAN DER WALT, R. GOMMERS, P. VIRTANEN, D. COURNAPEAU, E. WIESER, J. TAYLOR, S. BERG, N. J. SMITH, R. KERN, M. PICUS, S. HOYER, M. H. VAN KERKWIJK, M. BRETT, A. HALDANE, J. F. DEL RÍO, M. WIEBE, P. PETERSON, P. GÉRARD-MARCHANT, K. SHEPPARD, T. REDDY, W. WECKESSER, H. ABBASI, C. GOHLKE, and T. E. OLIPHANT, "Array programming with NumPy," *Nature*, **585**, *7825*, 357–362 (9 2020).
5. S. K. LAM, A. PITROU, and S. SEIBERT, "Numba: A LLVM-Based Python JIT Compiler," in "Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC," Association for Computing Machinery, New York, NY, USA (2015), LLVM '15.
6. T. G. MATTSON, T. A. ANDERSON, G. GEORGAKOUDIS, K. HINSEN, and A. DUBEY, "PyOMP: Multithreaded Parallel Programming in Python," *Computing in Science and Engineering*, **23**, *6*, 77–80 (11 2021).
7. L. ODEN, "Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing," in "Proceedings - 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2020," (2020), pp. 216–223.
8. F. D. WITHERDEN, A. M. FARRINGTON, and P. E. VINCENT, "PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, **185**, *11*, 3028–3040 (2014).
9. M. BAYER, "Mako: Templates for python," (2013).
10. F. D. WITHERDEN, "Python at petascale with PyFR or: how I learned to stop worrying and love the snake," *Computing in Science & Engineering*, **9615**, *c*, 1–1 (2021).
11. F. D. WITHERDEN, M. KLEMM, and P. VINCENT, "PyFR: Heterogeneous Computing on Mixed Unstructured Grids with Python," in "EuroSciPy," Cambridge (2015).
12. B. GANAPOL, R. BAKER, J. DAHL, and R. E. ALCOUFFE, "Homogeneous Infinite Media Time-Dependent Analytical Benchmarks," (2001).